RCP/CORBA Database Interface Review Code Overview

Organization of the Code

There are several packages associated with the RCP system. The RCP/CORBA client implementation all resides in one package, *rcp*. The interface for the server, used by the client, is defined in *rcp_db_server_idl*. The remainder of this document discusses only those parts of the *rcp* package most relevant to the RCP/CORBA client implementation.

Database classes

The class *RCPDatabaseServices* represents a single logical RCP database; this is the only part of the RCP database subsystem seen by the remainder of the RCP system.

The class *AbsRCPDatabase* is abstract; it defines the interface, used by *RCPDatabaseServices*, which must be implemented by any *concrete* database class.

The classes *FileSystemDB* and *RCPDatabaseInMemory* are two examples of concrete database classes; each implements the *AbsRCPDatabase* interface. *FileSystemDB* is the class that has been used for the past two years by DØ. It manipulates a simple database which is implemented purely in C++, with the persistent data being in the form of a set of human-readable (but not human-modifiable) files. *RCPDatabaseInMemory* exists primarily for testing purposes, but it is sometimes useful for users with special circumstances. It implements a database which has no persistence past the end of the program in which it is used.

The class *CORBA_RCPDatabase* implements the *AbsRCPDatabase* interface through use of the functions declared by the *rcp_db_server_idl* package. This class, and the classes and functions used to implement it, are the central topic for this review. As far as the RCP system (and clients thereof) are concerned, *CORBA_RCPDatabase* is part of a database server. As far as the *rcp_db_server_idl* interface is concerned, however, *CORBA_RCPDatabase* is the client of the CORBA server interface.

Translation Code

The RCP system normally works with a series of C++ classes: *RCPValue, RCPName, RCPID, etc.* The CORBA interface does not know about these classes. Because the client code may not run an ORB, the interface does not allow passing of any of these objects through the interface, to be used by the server.

Instead, the client code translates between the various C++ classes and the C++ representation of the CORBA IDL data structures which are passed through the IDL interface. Much of the bulk of the client code consists of this translation code.

More than one design tried

The translation code shows two different styles; this is a reflection of the author's learning process while writing the code.

My first attempt at producing the translation code was to write a few templates which would be able to handle most of the translation process, with only a few specialized functions. In the end, this did not work well, and the later code is less templated. The resulting mixture is a messy combination of a few templates and many template specializations.

There are several reasons why my templating experiment failed:

1. The CORBA C++ binding was not designed with support of generic programming in mind. For example, the CORBA C++ binding for *sequence* does not provide the sort of typedefs needed for reasonable generic programming. In consultation from Jim and Walter, we were able to come up with work-arounds for some of the problems, but the results are still ugly -- and thus not very maintainable. The CORBA string class is also not suitable for generic programming; it requires too many special memory handling procedures, making it more difficult to write generic code that deals with a CORBA string as easily as a built-in type.
2. The exception model implemented by Orbacus is not convenient for automatic handling. Ideally, a standardized library should exist to handle routine chores such as catching and handling exceptions thrown by the CORBA infrastructure (such as catching exceptions due to disconnection, and attempting reconnection, with user-configurable timeouts). Orbacus does not provide such a library.
3. The early definitions of some of the data structures in the *rcp_db_server_idl* package were not suitable for use in generic programming. This is now largely cured, but I have not tried to rework the code to produce a cleaner implementation.

As a much more minor note, some quirks in the template instantiation mechanism of KAI C++ v 4.0 (earlier versions did not manifest this problem) required explicit instantiation of a class template in specific source modules. As the code developed, and different functions were added to different source modules, the place where the explicit instantiation was required would change. Because this place had to be discovered by experiment -- a full clean and rebuild of the library and all test programs -- the process was quite time-consuming. This helped dissuade me from more extensive use of templates in this code.

Exception handling

Another way in which the code reflects my learning process is the style of error handling. In the earlier code, I was trying to make sure that neither ORB-generated exceptions nor Python-generated exceptions thrown across the CORBA connection could propagate to the user (of the RCP system). I still think this is the appropriate goal.

Since Orbacus does not provide a library to wrap calls to CORBA interface objects (in order to provide automatic handling of the ORB-related exceptions), and since there was none in existence at DØ, I spent some time trying to design such a library. However, there was no other interest in having such a library, and I was encouraged not to spend time producing such a library.

In the next generation of code I tried to be very careful with nested *try* and *catch* blocks, attempting to make sure no inappropriate exception propagated to the user. The resulting code is quite messy. In the final generation of code, I have almost entirely abandoned the laundering of exceptions, in the spirit of getting to a working product in less time.

Testing

The tests for the RCP system are fairly extensive; there are about 45 test programs in the *rcp* package. The most comprehensive part of the tests is run by a Perl script that manipulates the environment, making and deleting logical databases in a configuration necessary to tests all (I think) of the use cases of the RCP system. This is the suit of tests that is used to very that the *FileSystemDB* class is performing the correct tasks.

The tests for *CORBA_RCPDatabase* and the *rcp_db_server_idl* interface consist of a few tests to very the CORBA connectivity, a few tests to very the functioning of the CORBA utility functions used to support *CORBA_RCPDatabase*, and a test suite for *CORBA_RCPDatabase* itself that is nearly identical to that used for *FileSystemDB*. The last 1-2 months of development of RCP have mostly consisted of working through this suite of tests, and discovering and fixing the parts of the client or server or Oracle schema that failed to satisfy the tests.